# Lecture 8: Fast Linear Solvers (Part 1)

# Gaussian Elimination and LU Factorization

Solve

$$
\begin{aligned}
E_1:&\quad a_{11}x_1 + a_{12}x_2 + \cdots a_{1n}x_n = b_1\\
E_2:&\quad a_{21}x_1 + a_{22}x_2 + \cdots a_{2n}x_n = b_2\\
&\qquad\qquad\qquad \vdots\\
E_n:&\quad a_{n1}x_1 + a_{n2}x_2 + \cdots a_{nn}x_n = b_n
\end{aligned}
$$

for $x_1, x_2, \ldots, x_n$.

- Matrix form $A\boldsymbol{x} = \boldsymbol{b}$:

$$
\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}
$$

- Direct method for solving $A\boldsymbol{x} = \boldsymbol{b}$ is by computing **LU factorization** $A = LU$

  Where $L$ is lower triangular and $U$ is upper triangular.

Solve
$$\begin{aligned} x_1 + x_2 + 2x_3 &= 6 \\ 2x_2 + x_3 &= 4 \\ 2x_1 + x_2 + x_3 &= 7 \end{aligned}$$

$$\left[\begin{array}{ccc|c} 1 & 1 & 2 & 6 \\ 0 & 2 & 1 & 4 \\ 2 & 1 & 1 & 7 \end{array}\right]$$

$$\begin{matrix} l_{21} = 0; l_{31} = 2 \rightarrow \\ (E_3 - 2 * E_1) \rightarrow (E_3) \end{matrix} \left[\begin{array}{ccc|c} 1 & 1 & 2 & 6 \\ 0 & 2 & 1 & 4 \\ 0 & -1 & -3 & -5 \end{array}\right] \quad \begin{matrix} l_{32} = -0.5 \rightarrow \\ (E_3 + 0.5 * E_2) \rightarrow (E_3) \end{matrix} \left[\begin{array}{ccc|c} 1 & 1 & 2 & 6 \\ 0 & 2 & 1 & 4 \\ 0 & 0 & -\frac{5}{2} & -3 \end{array}\right]$$

**Theorem** If Gaussian elimination can be performed on the linear system $A\boldsymbol{x} = \boldsymbol{b}$ *without row interchange,* $A$ can be factored into the product of *lower triangular* matrix $L$ and *upper triangular* matrix $U$ as $A = LU$:

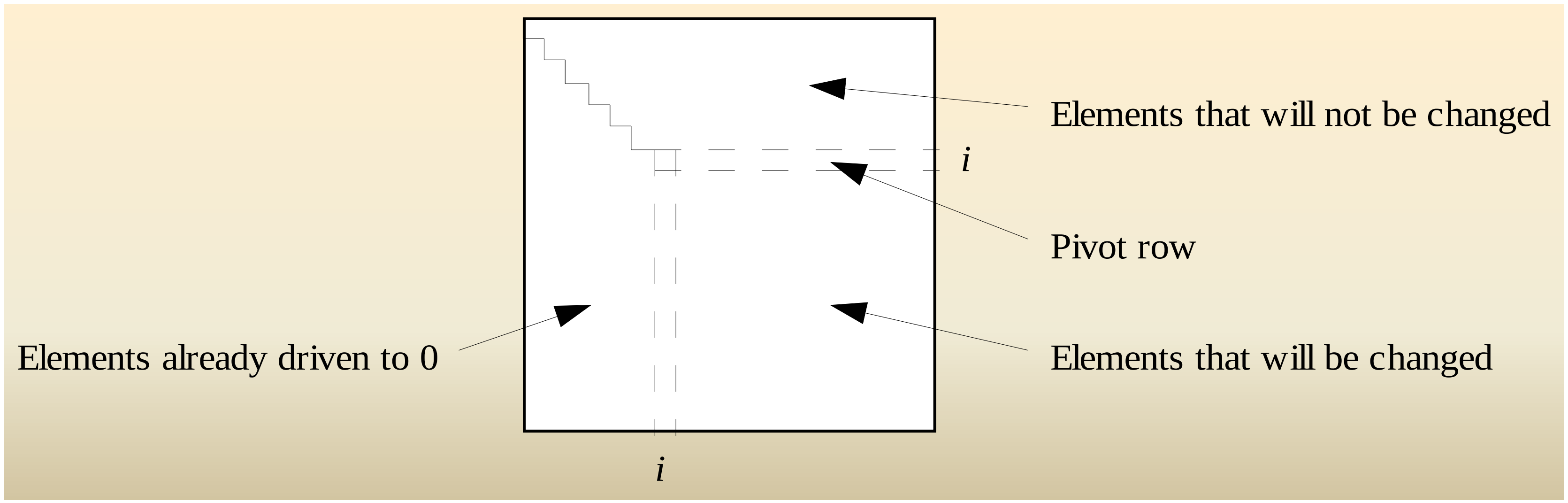
$$U = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \dots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \dots & a_{2n}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{nn}^{(n)} \end{bmatrix}, \qquad L = \begin{bmatrix} 1 & 0 & \dots & 0 \\ l_{21} & 1 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & \cdots & l_{n,n-1} & 1 \end{bmatrix}$$

1. $LU$ decomposition: $\mathrm{A} = LU$ so that $A\boldsymbol{x} = \boldsymbol{b}$ becomes

$$LU\boldsymbol{x} = \boldsymbol{b}$$

2. Solve $L\boldsymbol{y} = \boldsymbol{b}$ by forward substitution to obtain vector $\boldsymbol{y}$
3. Solve $U\boldsymbol{x} = \boldsymbol{y}$ backward for $\boldsymbol{x}$

# Iteration of Gaussian Elimination



# Gaussian Elimination Algorithm

- (n-1) stages of elimination are needed to obtain $U$. Assume all pivots at every stage are not 0.
- At the last stage, $U$ overwrites $A$.
- We assume that pivoting (row interchange) is not needed for simplicity.

```
for k = 1 to n − 1                    // loop over columns
  for i = k + 1 to n
    l_ik = a_ik / a_kk                 // multipliers for kth column
    b_i = b_i − l_ik b_k               // or multipliers for ith eqn.
  end;
  for j = k + 1 to n
    for i = k + 1 to n
      a_ij = a_ij − l_ik a_kj          // elimination step to remaining
    end;                               //  submatrix
  end;
end;
```

Gaussian elimination requires about $n^3/3$ paired additions and multiplications, and about $n^2/2$ divisions.

# Backward Substitution

After elimination, we obtain upper triangular $U\boldsymbol{x} = \boldsymbol{b}^{(n-1)}$.

**for** $k = n$ **to** $1$
    $x_k = b_k$
    **for** $i = k + 1$ **to** $n$
        $x_k = x_k - u_{ki}x_i$
    **end**;
    $x_k = x_k / u_{kk}$
**end**;

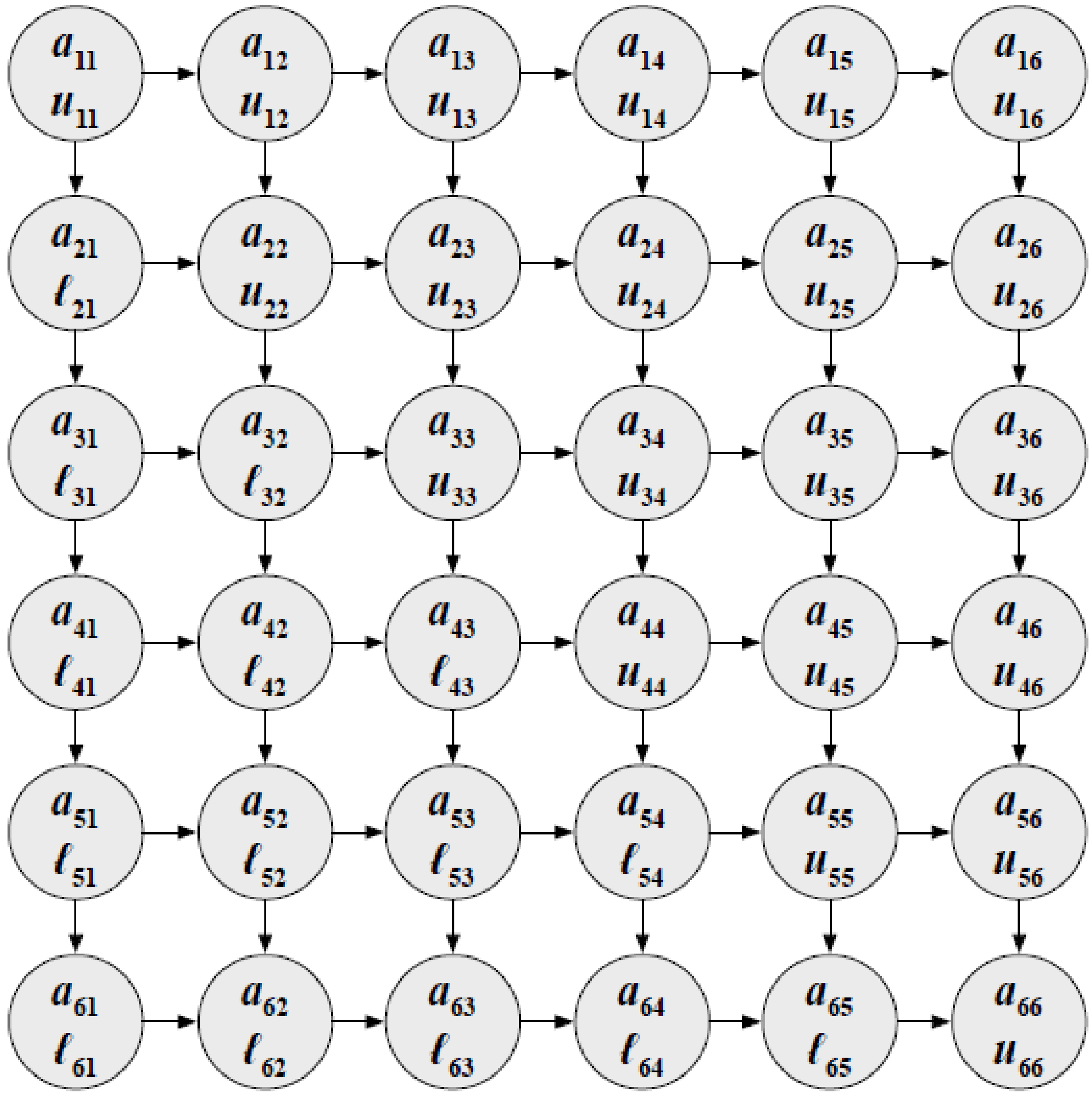
# Parallel LU Algorithm Design

- Assume a fine-grained decomposition, i.e., $a_{ij}$ is assigned to process $P_{ij}$.
- At the end of Computation, $P_{ij}$ stores

$$\begin{cases} u_{ij}, & if\ i \leq j \\ l_{ij}, & if\ i > j \end{cases}$$

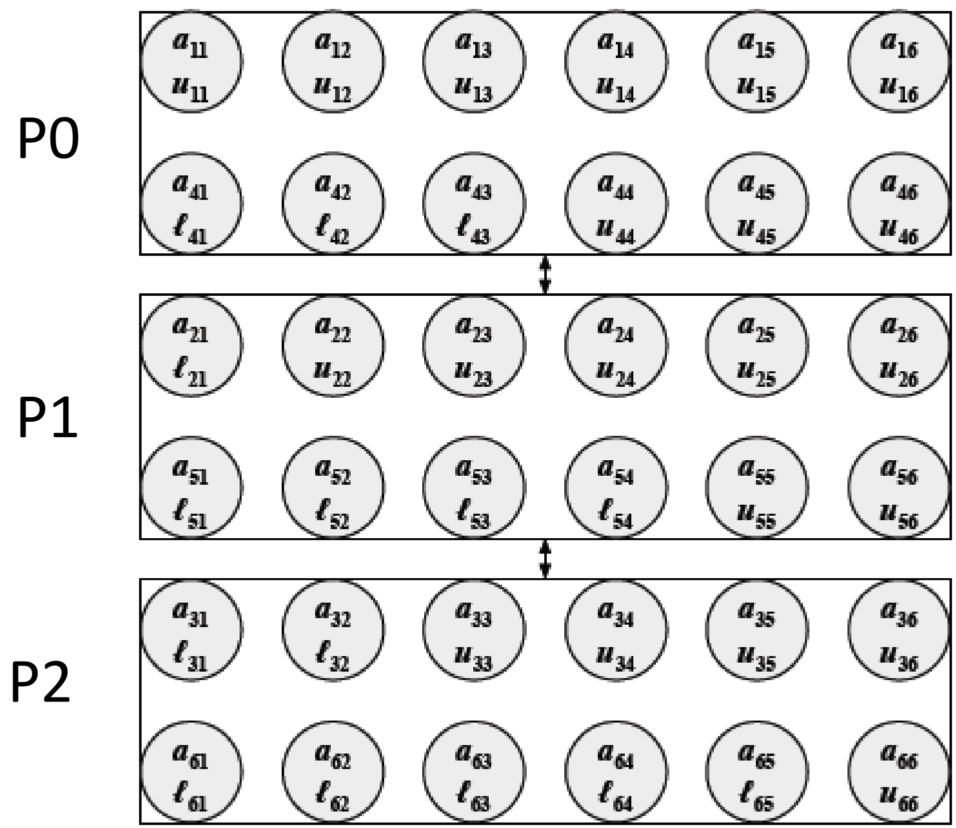
- **Outer** loop can not be executed in parallel; while the **inner** loop can be executed in parallel.
- Communications:
  - Broadcast row of $A$ vertically below
  - Broadcast $l_{ik}$ horizontally to tasks to right

# Fine-Grained Tasks and Communication



# Row-wise Cyclic Mapping Parallel Algorithm

- A few contiguous rows of $A$ (2 or 3 or more rows) are grouped into blocks. Distribute blocks to processes in a wraparound manner.
- Also associate corresponding elements of $\boldsymbol{b}$ and $\boldsymbol{x}$ of blocks to processes, respectively.



- Multipliers need not to be broadcasted horizontally, since any row of matrix is held entirely in one process.
- Vertical communications are still needed to broadcast a row of matrix to processes holding rows below it for updating.

# Row-wise Parallel Algorithm

**for** $k = 1$ **to** $n - 1$
  broadcast $kth$ row to processes holding $k + 1, \dots, n$ rows
  **for** processes holding $ith$ row, $i > k$,
    $l_{ik} = a_{ik}/a_{ii}$ // multipliers for $k$th column
  **end**;
  **for** processes holding $ith$ row, $i > k$
    **for** j $= k + 1$ **to** $n$
      $a_{ij} = a_{ij} - l_{ik}a_{kj}$ // elimination/update step
    **end**;
  **end**;
**end**;

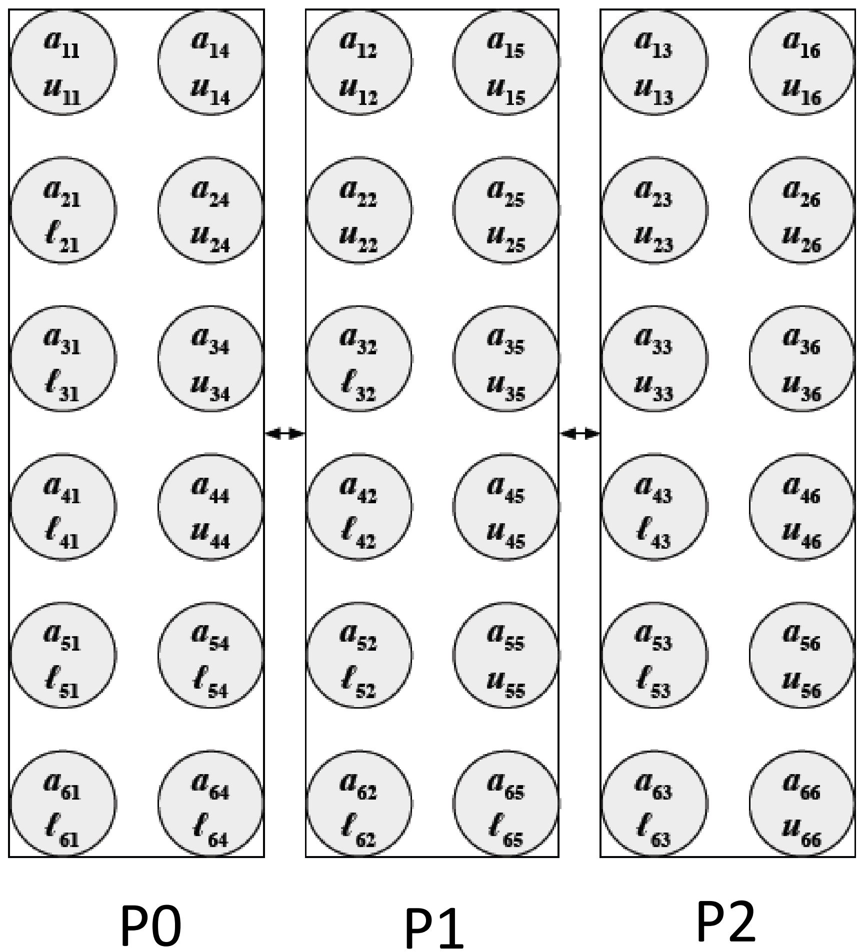
# Performance Analysis

Assume each row of matrix is assigned to a process.

- The inner loop at step $k$ involves $n-k$ multiplications and subtractions for processes holding $ith$ rows, $k<i<n$.
- At step $k$, there are $n-k$ divisions to compute multiplier ($\frac{a_{ik}}{a_{ii}}$)
- At step $k$, the one-to-all broadcast times time: $t_s + t_w(n-k)logn$
- Overall complexity: $t_c/p\sum_{k=1}^{n}(n-k)^2 + \sum_{k=1}^{n}(t_s + t_w(n-k)logn) \approx t_c n^3/(3p) + t_s nlogn + \frac{1}{2}n(n-1)t_w logn$

# Column-wise Cyclic Mapping Parallel Algorithm

- A few contiguous columns of $A$ (2 or 3 or more columns) are grouped into blocks. Distribute blocks to processes in a wraparound manner.

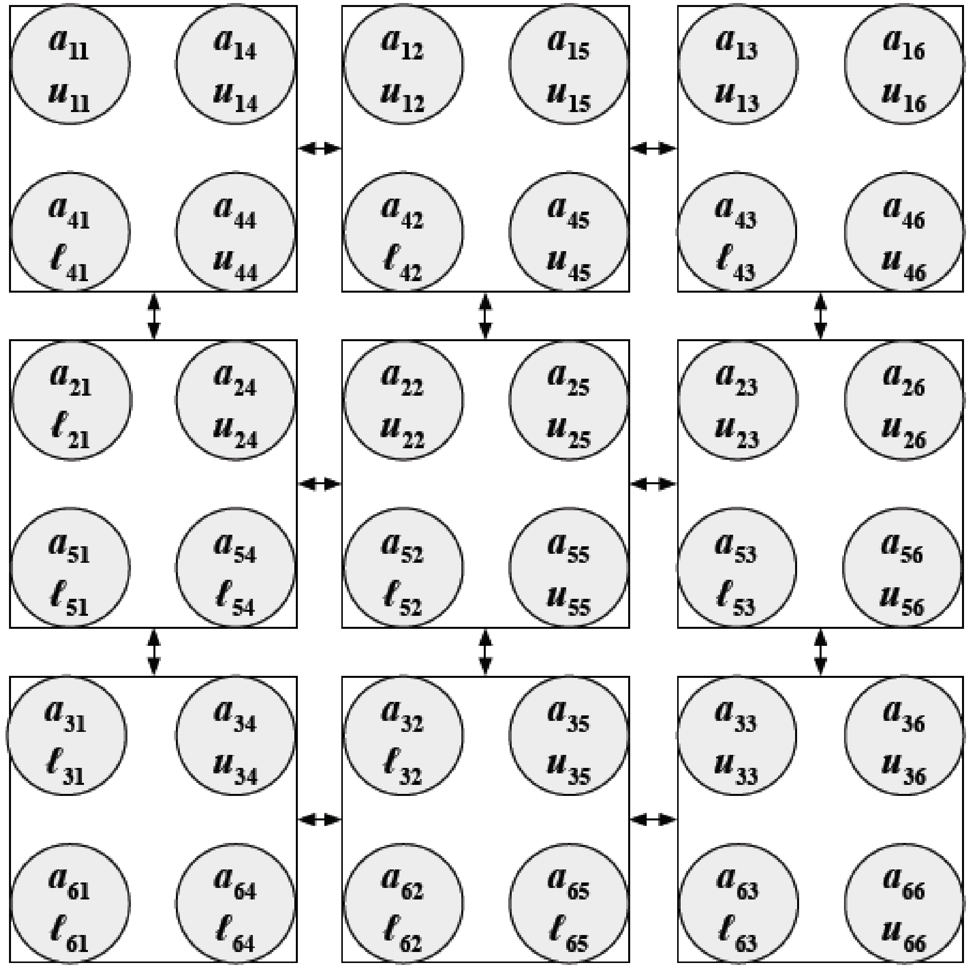


- Horizontal communications are needed to broadcast multipliers for updating.
- Vertical communications are not needed to broadcast a row of matrix, since any column is assigned to one process.

# Column-wise Parallel Algorithm

**for** $k = 1$ **to** $n - 1$
  **if** process holds $k$th column, **then**
    **for** $i = k + 1$ **to** $n$
      $l_{ik} = a_{ik}/a_{ii}$ // multipliers for $k$th column
    **endfor**;
  **endif**;
  broadcast $\{l_{ik} : k < i \leq n\}$ to processes holding $k, \dots, n$ columns
  **for** processes holds $j$th column, $j > k$
    **for** $i = k + 1$ **to** $n$
      $a_{ij} = a_{ij} - l_{ik}a_{kj}$ // elimination/update step
    **end**;
  **end**;
**end**;

# 2D Block Cyclic Mapping Parallel Algorithm



- With cyclic block mapping, each process holds several submatrices assembled globally. This improves both concurrency and load balance.
- Horizontal communications are needed to broadcast multipliers for updating.
- Vertical communications are also needed to broadcast a row of matrix, since any column is assigned to one process.

# 2D Block Cyclic Mapping Parallel Algorithm

```
for k = 1 to n − 1
    broadcast {a_kj : k ≤ j ≤ n} among columns of processes
    if process holds kth column, then
        for processes hold ith row, i > k
            l_ik = a_ik/a_ii                        // multipliers for kth column
        endfor;
     endif;
    broadcast {l_ik : k ≤ i ≤ n} to rows of processes
    for processes hold jth column,  j > k
        for processes hold ith row, i > k
            a_ij = a_ij − l_ik a_kj                 // elimination step
        end;
     end;
end;
```

# Gaussian Elimination with Partial Pivoting

- If pivot element $\approx 0$, significant round-off errors can occur.
- Partial pivoting finds the smallest $p \geq k$ such that $\left|a_{pk}^{(k)}\right| = \max_{k \leq i \leq n} |a_{ik}^{(k)}|$ and interchanges the rows $(E_k) \leftrightarrow (E_p)$.
- Partial pivoting is required for numerical stability of LU factorization and Gaussian elimination.

# Gaussian Elimination with Partial Pivoting Parallel Algorithm

- With 1D row algorithm or 2D block algorithm, searching pivot requires communication.
- With 1D column algorithm, searching pivot is local operation.
- Once pivot is found, index of pivot row must be communicated to all processes. Row interchange communication must be called.

# Pivot Searching

- Use MPI_Allreduce(), operator MPI_MAXLOC and derived data type MPI_DOUBLE_INT (struct {double, int}).

```
struct {
   double value;
   int    index;
} local, global;
...
local.value = fabs(a[j][i]);
local.index = j;
...
MPI_Allreduce (&local, &global, 1,
   MPI_DOUBLE_INT, MPI_MAXLOC,
   MPI_COMM_WORLD);
```

# Problems with These Algorithms

- All break parallel execution into computation and communication phases.
- Processes do not perform computations during the broadcast steps.
- As a result, communication time can be large enough to ensure poor scalability.
- Solution:  Pipelined communication and computation.